UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Relazione del progetto `chatterbox`

SOL a.a. 17-18

Autore:
Michele Zoncheddu

Ottobre 2018

Indice

1 Scelte progettuali

Di seguito verranno illustrate le più importanti scelte progettuali effettuate durante la realizzazione del progetto.

1.1 Strutture dati utilizzate

Le principali strutture dati utilizzate sono:

- coda concorrente (per il dispatch delle richieste);
- tabella hash concorrente (per utenti e gruppi);
- array circolare (per la history di ogni utente);
- array concorrenti:
  - per gli utenti online;
  - per le informazioni dei gruppi.

1.1.1 Coda concorrente

Serve a permettere la comunicazione tra thread `listener` e thread `worker` del pool. Non ha un limite di lunghezza, è realizzata come una lista, e la sincronizzazione è garantita da una mutex e da una variabile di condizione per l'attesa sulla coda vuota.
Il suo ruolo è anche quello di implementare il protocollo di terminazione.

1.1.2 Tabella hash concorrente

È la struttura dati più grande del server: serve a memorizzare tutti gli utenti e i gruppi registrati. È realizzata col metodo delle liste di trabocco, ed è possibile deciderne la dimensione a runtime.
Il server dev'essere in grado di gestire agevolmente un numero di utenti dell'ordine di 10.000, quindi utilizzare una variabile di mutua esclusione per ogni entry è troppo dispendioso: la tabella è stata divisa in blocchi, ognuno dei quali è protetto da una mutex. Il numero di blocchi è lo stesso del numero dei thread del pool, cosicché nel caso in cui tutti debbano accedere alla tabella contemporaneamente, possano farlo, distribuzione delle chiavi permettendo.
Ogni elemento della tabella hash ha al suo interno anche la history dell'utente: l'invio di un messaggio richiede necessariamente la ricerca del destinatario nell'insieme degli utenti registrati e il conseguente inserimento del messaggio nella history; le due strutture dati hanno quindi una dimensione e un numero di accessi molto simili, per questo motivo ho deciso di unificarle.

1.1.3 Array circolare

È la struttura dati che conserva la history degli ultimi messaggi di ogni utente. La dimensione è determinabile a runtime. Per poter ottenere statistiche precise, ogni messaggio è dotato di un flag che indica l'avvenuta consegna: è indispensabile per poter ottenere statistiche corrette nel caso di operazioni multiple di ottenimento della history (come avviene nello `stress test`), perché altrimenti il numero di messaggi non inviati diventerebbe negativo.

1.1.4 Array degli utenti online

Per ogni utente online è necessario memorizzare il file descriptor, e bisogna scrivere verso di essi in mutua esclusione, per evitare l'invio di più messaggi spezzettati.
Data la differenza di un ordine di grandezza rispetto al numero di utenti registrati, ho scelto di utilizzare un array (di dimensione determinabile a runtime), nel quale per ogni utente memorizzo nickname, fd e mutex associata. L'accesso all'array è protetto da una mutex a parte, e l'utilizzo combinato di quest'ultima con quelle degli utenti garantisce la scrittura in mutua esclusione verso ogni utente, senza ridurre il grado di parallelismo sulla struttura online.

1.1.5 Array dei gruppi

Per tenere traccia delle informazioni di ogni gruppo, ho deciso di utilizzare un array di strutture, ognuna delle quali contiene, oltre al nome del gruppo, il nome del suo creatore, il numero di partecipanti e la lista di partecipanti. L'intero array è protetto da una sola mutex, per via della brevità delle operazioni.

1.2 Strutturazione del codice

Tutte le funzioni, strutture dati di ogni file e punti critici sono stati adeguatamente commentati: è possibile ottenere una pagina web interattiva (`html/index.html`) eseguendo il comando `doxygen` all'interno della cartella del progetto.

Per il controllo dei valori restituiti da chiamate di sistema, chiamate di libreria, chiamate di funzione e `malloc` il cui fallimento implica la terminazione del server, sono state implementate rispettivamente le macro `SYSCALL`, `LIBCALL`, `FUNCALL` e `MALLOC`, che terminano il processo e stampano un messaggio di errore personalizzato per facilitare il debugging.

1.2.1 Suddivisione in file

Il core del server risiede nel file `chatty.c`: contiene il metodo `main`, che si occupa di invocare il parser, creare le strutture dati, mascherare i segnali, creare tutti i thread ed aspettare la loro terminazione; sono anche presenti i metodi `sigHandler`, `listener` e `worker`, i thread che si occupano rispettivamente di

gestire i segnali, le richieste di connessione dei client e le risposte ai client.
Le implementazioni di tutte le funzioni utilizzate dai `worker` sono all'interno del file `operations.c`.
La gestione delle strutture per gli utenti registrati, utenti online, gruppi e coda concorrente sono delegate rispettivamente ai file `users`, `online`, `groups` e `queue`.
Il file `util.h` contiene macro e funzioni di utilità.
I restanti file sono stati forniti dai docenti e adattati dallo studente.

1.2.2 Librerie

Tutte le funzioni e strutture dati dei file `connections`, `queue`, `users`, `online`, `operations` e `groups` vengono inserite all'interno della libreria statica `libchatty.a`.

2 Schema di funzionamento e interazione tra processi e thread

Di seguito uno schema che mostra in maniera molto semplificata il funzionamento del server, con una breve spiegazione.

2.1 Spiegazione dello schema: interazione `listener/worker`

Le richieste dei client verso il server vengono intercettate dal thread `listener` (*blu*), che si occupa di inserirle nella coda concorrente (*verde scuro*). Dall'altro capo della coda ci sono i thread `worker` del pool (*arancione*), estrarranno le

richieste e le gestiranno, inoltrando eventuali messaggi (*giallo*) ai destinatari. I thread del pool, per comunicare al listener che hanno terminato di servire un client, utilizzano una `pipe` (*verde chiaro*).
Richieste multiple da uno stesso client vengono serializzate, ignorando quelle pendenti fino a quando non è stata terminata la gestione della richiesta precedente. I thread `worker` non comunicano mai tra di loro.

2.2 Gestione dei segnali

La gestione dei segnali è delegata al thread `signalHandler`: si mette in costante attesa di segnali, impostando il flag per la terminazione nel caso il segnale intercettato sia `SIGINT`, `SIGTERM` o `SIGQUIT`, o quello per la stampa delle statistiche nel caso il segnale sia `SIGUSR1`.

2.3 Protocollo di terminazione

Questo protocollo viene implementato attraverso la coda concorrente, nella quale il thread `listener`, quando rileva che è stato settato il flag di terminazione, manda un messaggio speciale (identificato con la macro `END`) ad ogni thread del pool; quando il `main` riuscirà a fare le join di tutti i thread, effettuerà la fase finale di cleanup e terminerà.

3 Parti opzionali e aggiuntive

Sono state implementate tutte le funzioni sui gruppi, compresa la cancellazione di un gruppo, che può essere richiesta:

- dal creatore del gruppo, se ancora presente nel gruppo;
- dal primo utente chi si è aggiunto al gruppo dopo il creatore, se quest'ultimo ha abbandonato il gruppo (così via ricorsivamente, fino a quando il gruppo non ha nessun membro, e viene cancellato).

È possibile eseguire il test sui gruppi col comando `make test6` all'interno della cartella del progetto.

Sono stati inoltre aggiunti tre parametri di configurazione nei file `chatty.conf*`:

- `MaxUsers`: numero massimo **consigliato** di utenti registrati. Si può eccedere questo valore, a costo di perdere efficienza nelle operazioni che agiscono sulla tabella hash.
  Valore di default: `32768`;
- `MaxOnlineUsers`: numero massimo di utenti online.
  Valore di default: `4096`;
- `MaxGroups`: numero massimo di gruppi totali.
  Valore di default: `1024`.

4 Testing

Il progetto è stato testato con successo nelle seguenti configurazioni di sistema:

- Ubuntu 16.04 e Windows 10 su `Intel Core i7-2670QM`;
- MacOS 10.14 su `Intel Core i5-5257U`;
- Ubuntu 18.04 su `Intel Core i5-7400`.

5 Possibili miglioramenti

- Al posto dello `switch-case`, in caso di crescita del numero di operazioni, sarebbe molto più efficiente utilizzare nel `worker` una tabella hash con chiave l'operazione richiesta dal client, e come valore il puntatore all'handler di `operations.c` associato.
  Esempio: chiave: `REGISTER_OP`, valore: puntatore alla funzione `registerOp`;
- Mantenere la history per i gruppi (non implementata perché il client fornito non ha operazioni per richiederla);
- Persistenza degli utenti e dei gruppi registrati al riavvio del server;
- Memorizzazione di data e ora di invio e ricezione dei messaggi.